SpaceWire as a Cube-Sat Instrument Interface

Missions and Applications – Long

Susan C. Clancy, Matthew D. Chase, Anusha Yarlagadda, Michael D. Starch, James P. Lux
Jet Propulsion Laboratory, California Institute of Technology
Pasadena, CA, USA
susan.c.clancy@jpl.nasa.gov

***Abstract*— SpaceWire is used in the control and data interface for an instrument on a pair of small satellites, one of which was launched in summer 2017. The instrument SpaceWire interface is implemented in a Field Programmable Gate Array as an instantiated core controlled by a LEON3FT CPU, which is also implemented as an instantiated core. The UT699 processor in the flight computer provides the spacecraft side's SpaceWire interface.**

A simple message based protocol consisting of four message types was defined, based on existing SpaceWire standards. One was for passing commands to and responses from the instrument in the form of text strings similar to those from a system console where each line of text is passed in a SpaceWire message. Another was for passing spacecraft time to the instrument. The third was for transferring files using a subset of the Remote Memory Access Protocol (RMAP). The fourth was for retrieving science data from the instrument.

A set of user application programming interface (API) routines provided an abstracted interface to both the serial console (used during debug) and the SpaceWire device interface.

Early instrument development and testing was done with a set of utilities that controlled a Star-Dundee USB-SpaceWire brick providing a user interface similar to a serial console terminal emulator with the addition of file and data transfers. Later in the integration and test process, these utilities were integrated with the COSMOS ground systems software used for spacecraft control, providing a seamless transition from standalone instrument tests to benchtop flat-sat test and full spacecraft level tests.

Keywords—cubesat; SpaceWire; LEON3FT; RMAP; COSMOS; USB-Spacewire brick

I. Introduction

SpaceWire is used in the command and data interface between an instrument payload and the host spacecraft. The host spacecraft is a standard 3U (approximately 30cm x10cm x10cm) spacecraft from Space Dynamics Laboratory in Logan, UT. The spacecraft provides the support infrastructure: solar panels, batteries, flight computer, GPS receiver, and attitude control. The instrument is based on the JPL Iris radio [1] which uses a Xilinx Virtex 6 Field Programmable Gate Array (FPGA) that serves both as a digital signal processor and as the instrument controller. The instrument runs the RTEMS operating system [2] on an instantiated LEON3FT CPU core. SpaceWire is used to pass commands to the spacecraft as ordinary human readable strings, and the spacecraft sends back telemetry, also as human readable strings.

The instrument is commanded to collect data for about 10 minutes at a user specified time. The digital signal processing algorithms produce data at about 5 Mbps which is stored in flash memory in the instrument. The ground operators command the spacecraft to requests the science data, which is streamed out to the spacecraft at high speed over the SpaceWire interface where it is stored in spacecraft flash memory. Ultimately, the spacecraft sends the data to a ground station when requested.

There are provisions for the spacecraft to transfer files to and from the instrument using a subset of the RMAP protocol [3]. In addition, the instrument receives periodic time updates passed from the onboard spacecraft GPS receiver once a second, so that the time of capture can be set accurately.

II. Physical Implementation

The SpaceWire interface on the instrument side was implemented using the Gaisler GRSpW2 core instantiated in the

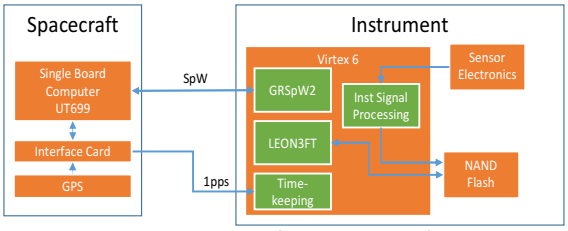

Figure II-1 Spacecraft-Instrument Interfaces

Virtex 6 FPGA, along with the LEON3FT CPU core. The 4 LVDS pairs are directly connected to the pins with no external driver or receiver to minimize changes to the existing processor board design. The spacecraft Single Board Computer (SBC) [4,5] uses the UT699 CPU, which includes the SpaceWire interface as part of the chip. CubeSat's are physically small and compact, Omnetics NanoD (MIL-DTL-32139) connectors were used throughout the spacecraft, with twisted pair wires for the 5 cm cable. The SpaceWire interface used a 9 pin configuration, with the pin assignment identical to the usual SpaceWire Micro-D.

III. Protocol

The interface between the instrument and the spacecraft uses four different variable length packet formats, each identified by a unique protocol identifier (PID), shown in Table III-1. The use of different PIDs allows easy separation of the messages

when received, without needing to further parse the message to determine the content. For the most part, the spacecraft Flight Software (FSW) just passes the entire SpaceWire message through unchanged in either direction. The spacecraft command header is stripped off and the embedded SpaceWire command message is passed through to the instrument. The protocols described below are those processed by the instrument.

Table III-1 Protocol Ids

PID	PROTOCOL ID DESCRIPTION
0x01	RMAP – used for file and binary data transfer to/from the instrument
0xF0	text (ASCII) data to from the instrument (stdin, stdout)
0xF1	Sampled Data as VITA-49 packets returned from the instrument
0xF2	GPS Binary message to the instrument

The SpaceWire packets follow the format defined in the SpaceWire Specification [6]. The packet includes a destination address, payload of application specific data, and an End of Packet (EOP) marker as shown in Figure III-1.

For an instrument SpaceWire packet, the address was always 0xFE, regardless of whether it was sent to or received from the instrument since it was a point to point link. The instrument includes the Protocol Id (PID) which identifies the packet type, followed by the payload data, and ending with the Cyclic Redundancy Check (CRC) used to detect data corruption.

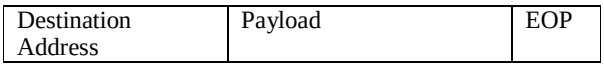

Figure III-1 SpaceWire Packet Format

Figure III-2 Packet Format

A. Serial Console Emulation

The instrument software provides a serial console style interface with variable length text commands which produce variable length text response messages. These command and response messages are identified using the 0xF0 Protocol Id. The API routine used by the software to send text messages inserts a text sequence number and the instrument time before each message.

Figure III-3 Text Packet Format

B. File Transfer To the Instrument

File transfers to and from the spacecraft use a subset of the existing RMAP protocol. These messages use the 0x01 Protocol Id and the RMAP format which includes the destination

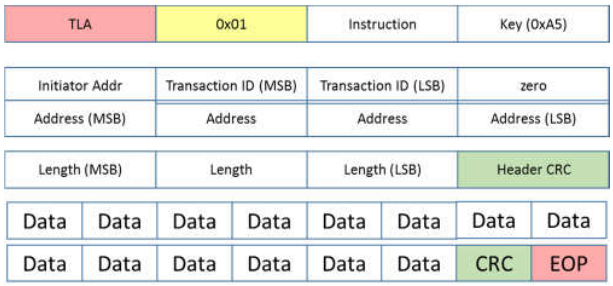

Figure III-4 RMAP Write Format (data to instrument)

target logical address (TLA), RMAP header fields, data, RMAP CRC, and end of packet marker. The RMAP header fields identify the RMAP instruction, key, initiator address, transaction id, address, length as shown in Figure III.4. The address fields define the position within the file being transferred and the length is the length of the data portion within the message. Write Reply messages are returned to the spacecraft with the format given in Figure III-5 if a Write-With-Reply instead of a Write-Without-Reply instruction is used.

Figure III-5 - Write Reply (from instrument, if requested)

To send a file, an FWRITE command is sent giving the name of the file and the maximum length of the file. The file length allows allocation of the space in the in-memory file system in advance of the transfer. Then all the RMAP Write packets are sent with the file contents. The payload data from each RMAP Write message is written to the specified file at the address in the message header. After all the RMAP data messages are sent, an FCLOSE command is sent. RMAP messages received when there is no active FWRITE are discarded with an error message.

C. File Transfer From the Instrument

Files are transferred from the instrument to the spacecraft by using RMAP Read messages as shown in Figure III-6. An FREAD command is sent to the instrument, and the instrument replies with a series of RMAP Read messages containing the file contents. The Transaction Id is used a s a sequence number and used to detect missing messages by the ground processing.

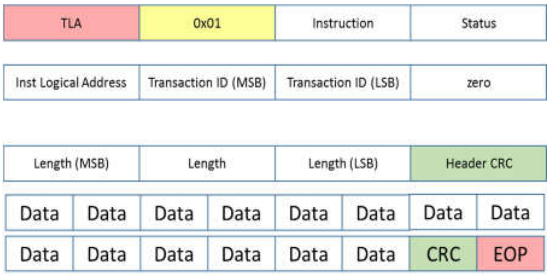

Figure III-6 - RMAP Read (data from instrument)

D. Science Data Transfer from the Instrument

The primary science output of the instrument is long streams of sampled data representing the received sensor signals. The data is encoded in the Virtual Radio Transport (VRT) packets defined in the VITA-49 specification [7]. A SpaceWire message was defined that contained the entire VRT packet with the 0xF1 protocol ID. A VRT packet stream consists of periodic context packets interspersed in a stream of data packets, each containing 250 samples. The context packet that precedes the data packets identifies information about the data stream.

Figure III-7 VITA-49 Data Packet

The command which starts sending science data allows a pacing delay to be inserted between each message, typically on the order of 10 milliseconds, to limit the overall data rate to the spacecraft.

E. GPS Time to the Instrument

The spacecraft uses a GPS receiver to determine the time, which is provided to the instrument via a GPS message sent as a 0xF2 Protocol Id message. The original design of the instrument used the time distribution message using a CCSDS Unsegmented Time proposed by Habinc, *et al.* [8]. However, to minimize the changes in the existing spacecraft flight software, a simpler message that encoded the GPS week and milliseconds was used, as shown in Figure III-8.

Figure III-8 GPS Packet Format

The GPS milliseconds represents the number of milliseconds that have elapsed between the GPS epoch and the next GPS one-pulse-per-second tick (1PPS), which is received on a discrete input line. As each GPS message is processed, the given GPS time is saved and the system clock ticks are stored with the associated 1PPS tick. The instrument internal clock is latched on each 1pps, along with the last received GPS Milliseconds value, which is used to calculate GPS time from instrument internal clock time. Software logic detects missing 1pps pulses or GPS time messages that are out of sequence.

IV. SpaceWire API

The instrument software implements a simple message passing style API to provide a consistent interface to the SpaceWire hardware. Transmit and receive tasks handle data sent or received and the GAISLER SPW2 SpaceWire device driver library functions perform the low-level device operations with the hardware. The "spacewire_init" function combines some of the low-level function calls needed to initialize the hardware into a higher level API function call.

The API functions (Table IV-1) format data sent from the instrument into a SpaceWire message and queue it as a transmit request to the transmit queue (TxQ). An instrument user application sending data calls one of the three API "send" functions to send data as text (PID=F0), RMAP data (PID=01), or VITA-49 VRT data (PID=F1). The transmit task (See Figure IV-1) dequeues each TxQ entry and calls the SPW2 spw_tx() and spw_checktx() device driver functions. The spw_tx() function transmits the packet out over SpaceWire. The spw_checktx() function blocks until the packet is transmitted or fails with an error. The transmit success and fail statistics are updated and can be captured and reported by the "spacewire_status" command.

Table III-1 SpaceWire API

FUNCTION NAME	DESCRIPTION
spacewire_init()	Initialize and sets up the buffer for sending and receiving data
send_data_packet(len,tid,buf)	Send an RMAP data packet
send_text_packet(len,buf)	Send a text packet
send_vita49_packet(len,buf)	Send VITA-49 packet
send_write_reply_packet(len, id,buf)	Send fwrite reply
set_fwrite_params(fn, fsize)	Updates file IO name and size from the FWRITE command
write_packet_to_file(pkt)	Decodes an RMAP packet and writes the data to a file
dump_packet(buf,len)	Outputs packet data as a series of text messages with HEX ASCII encoded data values
print_diag_packet(buf, len,opt)	Decodes and outputs the contents of a packet in a series of text messages

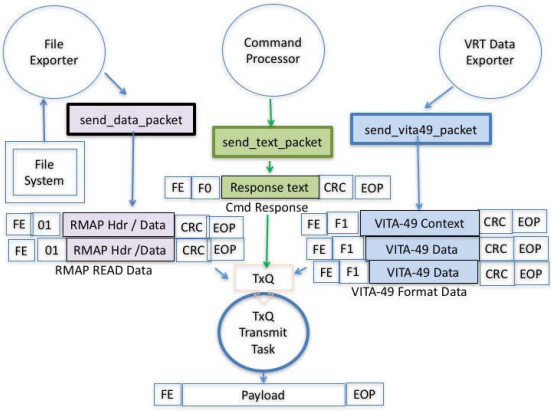

Figure IV-2 SpaceWire Transmit Software Architecture

The receive task (see Figure IV-2) handles three types of incoming packets which are identified by their PID. These are text commands (PID=0xF0), RMAP WRITE data (PID=0x01), or GPS data (PID=0xF2). The SPW2 spw_rx() and spw_check_rx() device driver functions are called within the receiver task to receive packets. The spw_rx() function sets up the input buffer to receive data and the check_rx() blocks until data arrives or detects a failure. The receive success and fail

statistics are updated and can be captured and reported by the "spacewire_status" command.

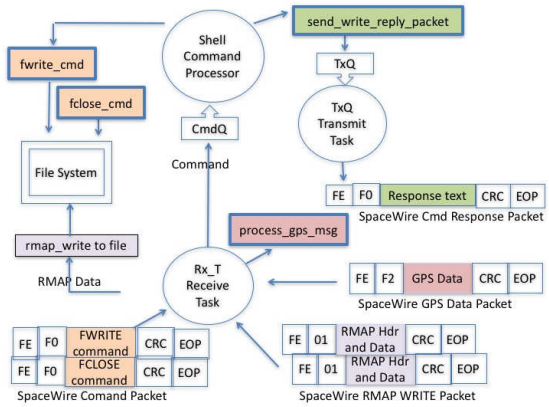

Figure IV-2 SpaceWire Receive Software Architecture

Incoming text messages are forwarded to the command input message queue which is serviced by the command processor. Any incoming RMAP WRITE data packets are decoded and the data is written to the file previously opened by an FWRITE command. Any incoming GPS data packets are processed by calling the GPS API.

The integrity of the incoming SpaceWire messages is checked using the data CRC and, if it matches the expected CRC value, the message is handled. If not, a receive error count is incremented and the message is discarded.

There are 4 API entry points shown in Table IV-1 which map to specific commands that the instrument can receive: fwrite() and fclose() used before and after importing data into a file, a "configure" command to enable diagnostic messages, and a "status" command to report SpaceWire API statistics.

Table IV-1 SpaceWire Related Commands API

FUNCTION	DESCRIPTION
fclose_cmd	Closes a file previously opened by the FWRITE command
fwrite_cmd	Creates a zero-filled file of the specified length in preparation for writing incoming RMAP data
spacewire_cfg_cmd	Configures SpaceWire API diagnostic option on or off; When diagnostics are on, the contents of sent and received messages is output for debugging
spacewire_status_cmd	Reports SpaceWire API statistics

V. Ground Support Software

The spacecraft ground system and much of the testing uses the open source COSMOS system from Ball Aerospace [9, 10].

The COSMOS system uses the Ruby language with procedures and modules written in Ruby to run test sequences, send commands, receive and format telemetry. During the development of the instrument, we had 3 fundamentally different use configurations:

- Test Bench Configuration - Standalone instrument on the bench with a Star-Dundee USB-SpaceWire brick
- FlatSat Configuration - Instrument connected to flatsat, using a RS422 link
- Spacecraft Configuration - Instrument integrated into the spacecraft

The Test Bench Configuration had a laptop computer with the Star-Dundee USB SpaceWire brick and associated tools and drivers connected via SpaceWire to the instrument. This configuration was used for the initial development and testing of the instrument hardware and software.

The FlatSat Configuration is a breadboard configuration including non-flight qualified instances of the spacecraft Single Board Computer and Interface Card. The spacecraft radio is emulated in the FlatSat by a serial port connected to a PC. The FlatSat includes a flight-like SpaceWire interface with an uplink / downlink radio interface simulated over an RS-232 connection.

The Spacecraft Configuration has the instrument integrated into the spacecraft and uses a ground station RF link for the uplink/downlink interface.

These configurations were accommodated by creating a Ruby software class hierarchy (see Figure V-1) that supported a standard (configuration independent) interface for interacting with the instrument. In this way, all command scripts, unit-tests, and other ground utilities operate with the instrument without

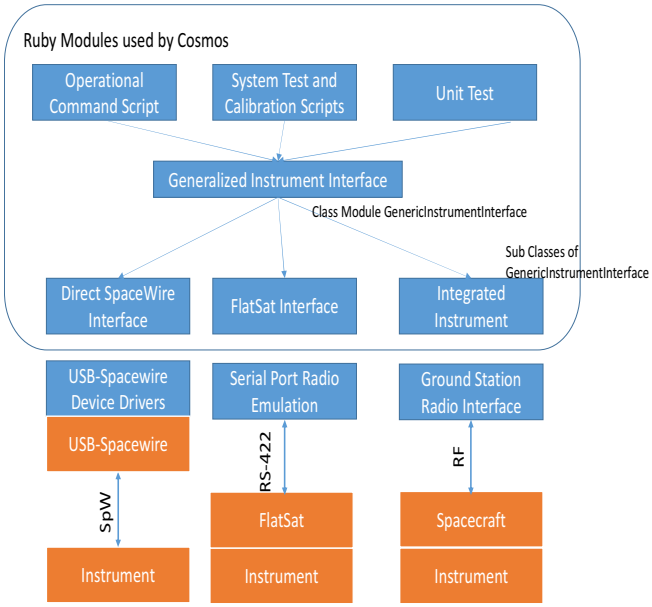

Figure V -1 Different configurations support a standard interface

needing to be altered when the physical configuration changed. Simply using a class's method, *e.g.* "send_inst_command" allows the tests to works on any interface or configuration. Ground processing software only needed to be written once to support all three testing configurations.

Ruby classes supporting the generic interface were created to automatically separate the incoming telemetry, a combination of instrument ASCII, file data, and VRT packets, into appropriate streams. These streams supported testing of the instrument and software in a flight-like environment.

ACKNOWLEDGMENT

This work was carried out at the Jet Propulsion Laboratory in Pasadena (JPL), California, under contract with the National Aeronautics and Space Administration. References herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the U.S. Government or the Jet Propulsion Laboratory.

REFERENCES

.

[1] Jet Propulsion Laboratory, "Iris V2.1 CubeSat Deep Space Transponder", https://www.jpl.nasa.gov/cubesat/pdf/Brochure_IrisV2.1_201611-URS_Approved_CL16-5469.pdf

[2] OAR Corporation, Real Time Executive for Multiprocessor Systems – RTEMS, https://www.rtems.org/

[3] European Cooperation for Space Standardization, "Space Engineering; SpaceWire Links, nodes, routers and networks,", ECSS-E-ST-50-12C, July 2008

[4] Space Dynamics Laboratory, "PEARL Spacecraft Platform", http://www.sdl.usu.edu/downloads/pearl.pdf, downloaded 25 Feb 2018.

[5] Q.Young, R. Burt, M. Watson, L. Zollinger "PEARL CubeSat Bus-Building Toward Operational Missions", Cubesat Summer Workshop -8-9 August 2009, Cal Poly San Luis

[6] European Cooperation for Space Standardization, "Space Engineering; SpaceWire Links, nodes, routers and networks,", ECSS-E-ST-50-11F, Dec 2006

[7] ANSI, "ANSI/VITA 49.0-2015 VITA Radio Transport Standard", https://shop.vita.com/ANSI-VITA-490-2015-VITA-Radio-Transport-VRT-Standard-AV490.htm

[8] S. Habinc, A. Sakthivel, M. Suess, "SpaceWire – Time Distribution Protocol", 2013 International Spacewire Conference.

[9] R. Melton, "Ball Aerospace COSMOS open source command and control system," Small Satellite Conference, August 2016.

[10] Ball Aerospace, "COSMOS – The user interface for command and control of embedded systems," http://cosmosrb.com/, retrieved 25 Feb.